APPENDIX A

```
—————————————————————————————————————————————————————————
—— TL_PARSER:  Procedure to perform statement-level parsing of Timeliner input.  ——
—————————————————————————————————————————————————————————

--- Modification History
---
--- 4/15/92    dee1441    Modified as required by new modularization...


------- EXTERIOR INTERFACES

--- TIMELINER SEED TYPES
    with tl_seed;   use tl_seed;

--- TIMELINER MASTER COMMON AREA
    with tl_common;   use tl_common;

--- TIMELINER COMPILATION-TIME COMMON AREA
    with tl_comp_com;   use tl_comp_com;

--- SUBROUTINE TO ISSUE INITIALIZATION-TIME ERROR MESSAGES
    with tl_cusser;   use tl_cusser;

--- TIMELINER GENERAL-PURPOSE SUBROUTINES
    with tl_subs;   use tl_subs;

--- TIMELINER COMPILATION-TIME SUBROUTINES
    with tl_comp_subs;   use tl_comp_subs;

--- USER-SUPPLIED INFORMATION
    with tl_user_info;   use tl_user_info;

--- USER-SUPPLIED SUBROUTINES
    with tl_user_comp_subs;   use tl_user_comp_subs;

--- TEXT INPUT/OUTPUT PACKAGE
    with text_io;   use text_io;

--- TIMELINER INPUT/OUTPUT PACKAGE
    with tl_io;   use tl_io;


------- PACKAGE BODY

package body TL_PARSER is

--- SUBROUTINE TO OBTAIN A NEW STATEMENT
    procedure obtain_statement (level     : in natural;
                                stat_line : out stat_string_type;
                                stat_leng : out column_type;
                                stat_type : out comp_type_type;
                                next_type : out comp_type_type;
                                stat_num  : out stat_pointer_type;
                                comp_loc  : out comp_pointer_type);


-----------------------------------------------
-- STATEMENT PARSING PROCEDURE --
-----------------------------------------------

    procedure PARSE_STATEMENT (return_code : in out return_code_type) is


    ------- OUTPUTS OF OBTAIN_STATEMENT (CANNOT BE STATIC)

    --- TYPE OF ACCUMULATED INPUT STATEMENT
        stat_type : comp_type_type range start_of_input..direct_statement;

    --- TYPE OF NEXT ACCUMULATED INPUT STATEMENT
        next_type : comp_type_type range start_of_input..direct_statement;

    --- STATEMENT NUMBER
        stat_num : stat_pointer_type;

    --- RESERVED LOCATION IN COMPONENT DATA ARRAY
        comp_loc : comp_pointer_type;


    ------- SAVED BLOCK INFORMATION

    --- SAVED BLOCK NAME
        block_name_save : name_type := (1..max_name_length => ' ');

    --- SAVED BLOCK TYPE
        block_type_save : comp_type_type := unknown_line;

    --- SAVED BLOCK LINE NUMBER
        block_num_save : block_pointer_type := 0;

    --- SAVED BLOCK LOCATION IN COMPONENT DATA
        block_loc_save : comp_pointer_type := 0;


    ------- SAVED CONSTRUCT INFORMATION

    --- SAVED CONSTRUCT TYPE
        const_type_save : comp_type_type := unknown_line;
```

Do Not Print

DO
RWD

36

```
--- SAVED CONSTRUCT LINE NUMBER
    const_num_save : stat_pointer_type := 0;

--- SAVED CONSTRUCT LOCATION IN COMPONENT DATA
    const_loc_save : comp_pointer_type := 0;

--- LOCATION WHERE "BEFORE" OR "WITHIN" STATEMENT NUMBER SHOULD GO...
    const_mod_loc : comp_pointer_type := 0;

--- LOCATION WHERE "OTHERWISE" STATEMENT NUMBER SHOULD GO...
    const_oth_loc : comp_pointer_type := 0;

--- LOCATION WHERE "END" STATEMENT NUMBER SHOULD GO...
    const_end_loc : comp_pointer_type := 0;

--- SAVED STATEMENT NUMBER OF "ELSE" STATEMENT
    const_else_num : stat_pointer_type := 0;


------- FOR CALLING COMPONENT PARSER

--- RESULTING COMPONENT TYPE
    ct, ct_left, ct_right : comp_type_type;

--- RESULTING COMPONENT LOCATION
    cp, cp_left, cp_right : comp_pointer_type;

--- RESULTING COMPONENT SIZE
    cs, cs_left, cs_right : comp_size_type;


------- MISCELLANEOUS

--- COLUMNS WHERE MATERIAL STARTS AND ENDS
    c0, c1 : column_type := 0;
    cop0, cop1 : column_type := 0;
    par0, par1 : column_type := 0;

--- COMPONENT POINTER
    loc : comp_pointer_type;

--- DUMMY NUMERIC
    num : scalar_double;

--- DUMMY BOOLEAN
    OK : boolean;

--- BLOCK NUMBER USED WHEN LOOKING FOR SEQS/SUBSEQS
    nb : block_pointer_type;

begin

--- INITIALIZE COUNTERS IF FIRST ENTRY...
    if stat_nest_level = 0 then
        n_names   := 0;
        n_blocks  := 0;
        n_stats   := 0;
        n_comps   := 1;
        n_cuss    := 0;
        n_ss_ops  := 0;
        n_bool_int_var := 0;
        n_num_int_var  := 0;
        n_char_int_var := 0;
        n_numeric_lits   := 0;
        n_character_lits := 0;
        trap_max_n_names           := 0;
        trap_max_statement_length  := 0;
        trap_max_block_nest_level  := 0;
        trap_max_stat_nest_level   := 0;
        trap_max_comp_nest_level   := 0;
        max_bool_buff_usage        := 0;
        max_num_buff_usage         := 0;
        max_char_buff_usage        := 0;
    end if;

--- INCREMENT NESTING LEVEL
    stat_nest_level := stat_nest_level + 1;

--- TRAP MAXIMUM STATEMENT NESTING DEPTH
    if stat_nest_level > trap_max_stat_nest_level then
        trap_max_stat_nest_level := stat_nest_level;
    end if;

--- COMPLAIN IF NESTING TOO DEEP
    if stat_nest_level > integer(max_stat_nest_level) then
        cuss (statement_nesting_too_deep, char(stat_nest_level));
    end if;

--- LOOP TO READ STATEMENTS
    stat_loop : loop

        --- OBTAIN A NEW STATEMENT
            obtain_statement (stat_nest_level, stat_line, stat_long,
                              stat_type, next_type, stat_num, comp_loc);

        --- DEBUG PRINT
            if print_level >= 5 then
                put_line ("from obtain_statement:");
```

37

```
            put_line ("    >" & stat_line(1..stat_leng) & '<');
        end if;

--- SET FIRST AND LAST COLUMNS TO EXCLUDE FIRST WORD
    c0 := word_break(1, stat_line);
    c1 := stat_leng;


    --------------------------------------
    --- UNPRODUCTIVE STATEMENT TYPES ---
    --------------------------------------

    if stat_type in unproductive_statements then

    --- COMPLAIN IF STATEMENT TYPE NOT RECOGNIZED
        if stat_type = unknown_line then
            cuss (statement_not_recognized);
        end if;


    --------------------------------------
    --- FUNCTIONAL STATEMENT TYPES ---
    --------------------------------------

    elsif stat_type in functional_statements then

    --- INDICATE IF IT'S TOO LATE FOR A DECLARE OR DEFINE STATEMENT
        if stat_type not in nonexecute_statements then
            defdecs_okay := false;
        end if;

    --- IF STATEMENT LIES OUTSIDE OF ANY SEQ OR SUBSEQ...
        if (current_block_type /= seq_blocker and
            current_block_type /= subseq_blocker) and
            stat_type not in block_openers and
            stat_type /= close_blocker and
            stat_type /= declare_statement and
            stat_type /= define_statement then
        --- COMPLAIN
            cuss (no_seq_or_subseq_open);
        end if;

    --- COMPLAIN IF LAST STATEMENT NOT A BLOCK CLOSER
        if next_type = end_of_input and stat_type /= close_blocker then
            cuss (end_with_close_blocker);
        end if;


    --------------------------------------
    ------- BLOCKING STATEMENTS -------
    --------------------------------------

        if stat_type in blocking_statements then

        --- IF IT'S A BLOCK OPENER...
            if stat_type in block_openers then

            --- IF THIS IS FIRST BLOCK OPENER...
                if current_block_type = unknown_line then
                --- IF IT'S A "BUNDLE" OPENER...
                    if stat_type = bundle_blocker then
                    --- PERMIT TWO LEVELS OF BLOCK NESTING
                        block_levels_allowed := 2;
                --- OTHERWISE ("SEQ" OR "SUBSEQ")...
                    else
                    --- ALLOW ONLY ONE LEVEL OF BLOCK NESTING
                        block_levels_allowed := 1;
                    end if;
                end if;

            --- SAVE BLOCK TYPE
                block_type_save    := stat_type;
                current_block_type := stat_type;
            --- SAVE BLOCK LOCATION
                block_loc_save := comp_loc;
            --- SAVE BLOCK NAME
                block_name_save := pad(wird(2, stat_line), max_name_length);

            --- COMPLAIN IF BLOCK NAME IS NULL...
                if trim(block_name_save) = "" then
                    cuss (block_not_named);
                end if;
            --- FILE BLOCK NAME
                file_name(block_name_save, stat_num,
                    comp_data(comp_loc+4), comp_data(comp_loc+5));

            --- FILE BLOCK
                file_block(block_name_save, block_loc_save, block_num_save);
            --- FILE BLOCK NUMBER
                comp_data(comp_loc+1) := block_num_save;
            --- FILE BLOCK FIRST LINE
                comp_data(comp_loc+2) := stat_num;

            --- ALLOW DECLARATIONS AND DEFINITIONS
                defdecs_okay := true;

            end if;
```

```
--- MATERIAL PARTICULAR TO SPECIFIC BLOCKING STATEMENTS
    case blocking_statements'(stat_type) is

    ------------------------------------------------------------
    --- BUNDLE                                              ---
    ---    comp_data(comp_loc)     -  bundle_blocker        ---
    ---    comp_data(comp_loc+1)   =  pointer to block entry ---
    ---    comp_data(comp_loc+2)   =  first statement in bundle ---
    ---    comp_data(comp_loc+3)   -  last statement in bundle  ---
    ---    comp_data(comp_loc+4)   =  pointer to start of name  ---
    ---    comp_data(comp_loc+5)   =  pointer to end of name    ---
    ------------------------------------------------------------

        when bundle_blocker =>

        --- SET SCRIPT NAME FOR "BUNDLE"
            script_name := pad(wird(2, stat_line), max_name_length);
        --- COMPLAIN IF "BUNDLE" STATEMENT NOT FIRST LINE IN SCRIPT
            if stat_num /= 1 then
                cuss (bundle_must_come_first);
            end if;

        --- COMPLAIN IF BUNDLE NESTED TOO DEEP
            if stat_nest_level > 1 then
                cuss (bundle_nested_too_deep);
            end if;

        --- COMPLAIN IF ANY ADDITIONAL MATERIAL ON LINE
            cuss_extraneous_material (word_break(2, stat_line), stat_line);


    ------------------------------------------------------------
    --- SEQUENCE                                            ---
    ---    comp_data(comp_loc)     -  seq_blocker           ---
    ---    comp_data(comp_loc+1)   =  pointer to block entry ---
    ---    comp_data(comp_loc+2)   =  first statement in seq ---
    ---    comp_data(comp_loc+3)   =  last statement in seq  ---
    ---    comp_data(comp_loc+4)   =  pointer to start of name ---
    ---    comp_data(comp_loc+5)   =  pointer to end of name   ---
    ---    comp_data(comp_loc+6)   =  initial status         ---
    ------------------------------------------------------------

        when seq_blocker =>

        --- COMPLAIN IF SEQ NESTED TOO DEEP
            if stat_nest_level > block_levels_allowed then
                cuss (seq_nested_too_deep, char(stat_nest_level));
            end if;

        --- SET SEQUENCE INITIAL STATUS
            if wird(3, stat_line) - "INACTIVE" then
                comp_data(comp_loc+6) :=
                    half_integer(block_status_type'pos(seq_inactive));
            else
                comp_data(comp_loc+6) :=
                    half_integer(block_status_type'pos(seq_active));
            end if;

        --- COMPLAIN IF ANY ADDITIONAL MATERIAL ON LINE
            cuss_extraneous_material (word_break(3, stat_line), stat_line);

        --- SNAPSHOT CURRENT NUMBER OF NAMES
            n_names_snap := n_names;


    ------------------------------------------------------------
    --- SUBSEQUENCE                                         ---
    ---    comp_data(comp_loc)     =  subseq_blocker        ---
    ---    comp_data(comp_loc+1)   =  pointer to name       ---
    ---    comp_data(comp_loc+2)   =  first statement in subseq ---
    ---    comp_data(comp_loc+3)   =  last statement in subseq  ---
    ---    comp_data(comp_loc+4)   =  pointer to start of name  ---
    ---    comp_data(comp_loc+5)   =  pointer to end of name    ---
    ------------------------------------------------------------

        when subseq_blocker => null;

        --- COMPLAIN IF SUBSEQ NESTED TOO DEEP
            if stat_nest_level > block_levels_allowed then
                cuss (subseq_nested_too_deep, char(stat_nest_level));
            end if;

        --- COMPLAIN IF ANY ADDITIONAL MATERIAL ON LINE
            cuss_extraneous_material (word_break(2, stat_line), stat_line);

        --- SNAPSHOT CURRENT NUMBER OF NAMES
            n_names_snap := n_names;


    ------------------------------------------------------------
    --- CLOSE                                               ---
    ---    comp_data(comp_loc)     -  close_blocker         ---
    ---    comp_data(comp_loc+1)   =  block pointer to current block ---
    ------------------------------------------------------------

        when close_blocker =>

        --- COMPLAIN IF ANY CONSTRUCT IS OPEN...
            if const_loc_save > 0 then
```

39

```
                                cuss (construct_open_at_close);
                            end if;

                    --- IF NO BLOCK IS OPEN...
                        if block_loc_save = 0 then
                        -- COMPLAIN
                            cuss (no_block_open_at_close);
                    --- OTHERWISE...
                        else
                        --- SAVE POINTER BACK TO BLOCK OPENER
                            comp_data(comp_loc+1) := block_num_save;
                        --- FILE BLOCK LAST LINE
                            comp_data(block_loc_save+3) := stat_num;
                        --- COMPLAIN IF OPTIONAL NAME DOESN'T MATCH
                            if wird(3, stat_line) /= "" and
                                wird(3, stat_line) /= trim(block_name_save) then
                                cuss (close_name_mismatch,
                                    wird(3, stat_line) & " versus " & trim(block_name_save));
                            end if;
                        end if;

                    --- COMPLAIN IF SECOND WORD ABSENT OR NOT RECOGNIZED
                        if  wird(2, stat_line) /= "BUNDLE" and
                            wird(2, stat_line) /= "ACTIVITY" and
                            wird(2, stat_line) /= "PROCEDURE" and
                            wird(2, stat_line) /= "SEQ" and
                            wird(2, stat_line) /= "SUBSEQ" and
                            wird(2, stat_line) /= "SEQUENCE" and
                            wird(2, stat_line) /= "SUBSEQUENCE" then
                            cuss (close_incomplete);
                        end if;

                    --- COMPLAIN IF THERE IS EXTRANEOUS MATERIAL
                        cuss_extraneous_material (word_break(3, stat_line), stat_line);

                    --- SHOULD IT BE A 'CLOSE BUNDLE'?
                        if block_type_save = bundle_blocker then

                        --- COMPLAIN IF CLOSE TYPE DOES NOT CORRESPOND
                            if  wird(2, stat_line) /= "BUNDLE" and
                                wird(2, stat_line) /= "ACTIVITY" and
                                wird(2, stat_line) /= "PROCEDURE" then
                                cuss (close_mismatched,
                                    wird(2, stat_line) & " versus " & "BUNDLE");
                            end if;

                    --- SHOULD IT BE A 'CLOSE SEQUENCE'?
                        elsif block_type_save = seq_blocker then

                        --- COMPLAIN IF CLOSE TYPE DOES NOT CORRESPOND
                            if  wird(2, stat_line) /= "SEQ" and
                                wird(2, stat_line) /= "SEQUENCE" then
                                cuss (close_mismatched,
                                    wird(2, stat_line) & " /= " & "SEQUENCE");
                            end if;

                        --- SAVE POINTER BACK TO BLOCK OPENER
                            comp_data(comp_loc+1) := block_num_save;

                        --- RESET CURRENT NUMBER OF NAMES
                            n_names := n_names_snap;

                    --- SHOULD IT BE A 'CLOSE SUBSEQUENCE'?
                        elsif block_type_save = subseq_blocker then

                        --- COMPLAIN IF CLOSE TYPE DOES NOT CORRESPOND
                            if  wird(2, stat_line) /= "SUBSEQ" and
                                wird(2, stat_line) /= "SUBSEQUENCE" then
                                cuss (close_mismatched,
                                    wird(2, stat_line) & " /= " & "SUBSEQUENCE");
                            end if;

                        --- SAVE POINTER BACK TO BLOCK OPENER STATEMENT
                            comp_data(comp_loc+1) := block_num_save;

                        --- RESET CURRENT NUMBER OF NAMES
                            n_names := n_names_snap;

                        end if;

                end case;


        --------------------------------------
        ------ CONTROL STATEMENTS ------
        --------------------------------------

            elsif stat_type in control_statements then

            -- IF IT'S A CONSTRUCT OPENER...
                if stat_type in construct_openers then
                --- SAVE CONSTRUCT TYPE
                    const_type_save := stat_type;
                --- SAVE CONSTRUCT LINE NUMBER
                    const_num_save := stat_num;
                --- SAVE CONSTRUCT LOCATION
                    const_loc_save := comp_loc;
                end if;
```

40

```
--- IF IT'S A CONSTRUCT OPENER OR MODIFIER...
    if stat_type in construct_openers or
       stat_type in construct_modifiers then
    -- REMOVE OPTIONAL "THEN" FROM THE END
       if wird(-1, stat_line) = "THEN" then
          cl := word_break(-1, stat_line);
       end if;
    end if;


--- MATERIAL PARTICULAR TO SPECIFIC CONTROL STATEMENTS
    case control_statements'(stat_type) is

    ---------------------------------------------------------------
    -- WHEN or WHEN/CONTINUE
    ---    comp_data(comp_loc)     -  when_statement (or when_cont_statement)  ---
    ---    comp_data(comp_loc+1)   -  construct/modifier line (0 if when_cont) ---
    ---    comp_data(comp_loc+2)   =  otherwise/end line (0 if when_cont)      ---
    ---    comp_data(comp_loc+3)   -  loc of singular boolean component        ---
    ---------------------------------------------------------------

       when when_statement | when_cont_statement =>

       --- SET CONSTRUCT/MODIFIER LINE
           comp_data(comp_loc+1) := stat_num;

       --- IF IT'S A WHEN/CONTINUE STATEMENT...
           if wird(-1, stat_line) - "CONTINUE" then
           --- RESET COLUMN POINTER
               cl := word_break(-1, stat_line);
           --- REMOVE "THEN" IF ANY
               if wird(-2, stat_line) = "THEN" then
                  cl := word_break(-2, stat_line);
               end if;
           --- RESET STATEMENT TYPE TO "WHEN/CONTINUE"
               stat_type := when_cont_statement;
               comp_data(comp_loc) := half_integer(comp_type_type'pos(when_cont_statement));
           --- RESET INDICATOR THAT A CONSTRUCT IS OPEN
               const_loc_save := 0;

       --- OTHERWISE...
           else
           --- SET LOCATION FOR POSSIBLE MODIFIER LINE
               const_mod_loc := comp_loc + 1;
           --- SET LOCATION FOR POSSIBLE "OTHERWISE" LINE
               const_oth_loc := comp_loc + 2;
           --- SET TENTATIVE LOCATION FOR "END" LINE
               const_end_loc := comp_loc + 2;
           end if;

       --- INVOKE COMPONENT PARSER TO FILE CONDITION
           parse_component(stat_line(c0..cl), ct, comp_data(comp_loc+3), cs);

       --- COMPLAIN IF COMPONENT NOT OF SINGULAR BOOLEAN TYPE
           if cs > 1 or not (ct in boolean_comps or ct - unknown_comp) then
              cuss (stat_needs_boolean_single, stat_line(c0..cl));
           end if;


    ---------------------------------------------------------------
    -- WHENEVER                                                  ---
    ---    comp_data(comp_loc)     -  whenever_statement          ---
    ---    comp_data(comp_loc+1)   -  construct/modifier line     ---
    ---    comp_data(comp_loc+2)   -  end line                    ---
    ---    comp_data(comp_loc+3)   -  loc of singular boolean component ---
    ---------------------------------------------------------------

       when whenever_statement =>

       --- SET CONSTRUCT/MODIFIER LINE
           comp_data(comp_loc+1) := stat_num;

       --- SET LOCATION FOR POSSIBLE MODIFIER LINE
           const_mod_loc := comp_loc + 1;

       --- SET LOCATION FOR "END" LINE
           const_end_loc := comp_loc + 2;

       --- INVOKE COMPONENT PARSER TO FILE CONDITION
           parse_component(stat_line(c0..cl), ct, comp_data(comp_loc+3), cs);

       --- COMPLAIN IF COMPONENT NOT OF SINGULAR BOOLEAN TYPE
           if cs > 1 or not (ct in boolean_comps or ct - unknown_comp) then
              cuss (stat_needs_boolean_single, stat_line(c0..cl));
           end if;


    ---------------------------------------------------------------
    -- EVERY                                                     ---
    ---    comp_data(comp_loc)     -  every_statement             ---
    ---    comp_data(comp_loc+1)   -  construct/modifier line     ---
    ---    comp_data(comp_loc+2)   -  end line                    ---
    ---    comp_data(comp_loc+3)   -  loc of singular numeric component ---
    ---------------------------------------------------------------

       when every_statement =>

       --- SET CONSTRUCT/MODIFIER LINE
           comp_data(comp_loc+1) := stat_num;
```

41

```
    --- SET LOCATION FOR POSSIBLE MODIFIER LINE
        const_mod_loc := comp_loc + 1;

    --- SET LOCATION FOR "END" LINE
        const_end_loc := comp_loc + 2;

    -- INVOKE COMPONENT PARSER TO FILE CONDITION
        parse_component(stat_line(c0..c1), ct, comp_data(comp_loc+3), cs);

    --- COMPLAIN IF COMPONENT NOT OF SINGULAR NUMERIC TYPE
        if cs > 1 or not (ct in numeric_comps or ct = unknown_comp) then
            cuss (stat_needs_numeric_single, stat_line(c0..c1));
        end if;


----------------------------------------------------------------
-- IF                                                         ---
--  comp_data(comp_loc)      = if_statement                   ---
--  comp_data(comp_loc+1)    = next else/end line             ---
--  comp_data(comp_loc+2)    = loc of singular boolean component ---
----------------------------------------------------------------

    when if_statement =>

    --- SET LOCATION FOR MODIFIER LINE
        const_mod_loc := comp_loc + 1;

    --- SET LOCATION FOR "END" LINE
        const_end_loc := comp_loc + 1;

    --- INVOKE COMPONENT PARSER TO FILE CONDITION
        parse_component(stat_line(c0..c1), ct, comp_data(comp_loc+2), cs);

    --- COMPLAIN IF COMPONENT NOT OF SINGULAR BOOLEAN TYPE
        if cs > 1 or not (ct in boolean_comps or ct = unknown_comp) then
            cuss (stat_needs_boolean_single, stat_line(c0..c1));
        end if;


----------------------------------------------------------------
--- BEFORE                                                    ---
---  comp_data(comp_loc)      - before_statement              ---
---  comp_data(comp_loc+1)    = loc of singular boolean component ---
----------------------------------------------------------------

    when before_statement => null;

    --- COMPLAIN IF NO "WHEN/WHENEVER/EVERY" OPEN
        if const_type_save /= when_statement and
            const_type_save /= whenever_statement and
            const_type_save /= every_statement then
            cuss (before_within_outside);

    --- OTHERWISE...
        else
        --- COMPLAIN IF THERE IS ALREADY A "BEFORE" OR "WITHIN"
            if comp_data(const_mod_loc) /= const_num_save then
                cuss (before_within_already);
        -- OR COMPLAIN IF "BEFORE" DOESN'T FOLLOW CONSTRUCT OPENER
            elsif stat_num /= const_num_save + 1 then
                cuss (before_within_misplaced);
            end if;
        -- SET MODIFIER LINE
            comp_data(const_mod_loc) := stat_num;
        --- INVOKE COMPONENT PARSER TO FILE CONDITION
            parse_component(stat_line(c0..c1), ct, comp_data(comp_loc+1), cs);
        -- COMPLAIN IF COMPONENT NOT OF SINGULAR BOOLEAN TYPE
            if cs > 1 or not (ct in boolean_comps or ct = unknown_comp) then
                cuss (stat_needs_boolean_single, stat_line(c0..c1));
            end if;
        end if;


----------------------------------------------------------------
--- WITHIN                                                    ---
---  comp_data(comp_loc)      = within_statement              ---
---  comp_data(comp_loc+1)    = loc of singular numeric component ---
----------------------------------------------------------------

    when within_statement => null;

    --- COMPLAIN IF NO "WHEN/WHENEVER/EVERY" OPEN
        if const_type_save /= when_statement and
            const_type_save /= whenever_statement and
            const_type_save /= every_statement then
            cuss (before_within_outside);

    --- OTHERWISE...
        else
        --- COMPLAIN IF THERE IS ALREADY A "BEFORE" OR "WITHIN"
            if comp_data(const_mod_loc) /= const_num_save then
                cuss (before_within_already);
        --- OR COMPLAIN IF "WITHIN" DOESN'T FOLLOW CONSTRUCT OPENER
            elsif stat_num /= const_num_save + 1 then
                cuss (before_within_misplaced);
            end if;
        --- SET MODIFIER LINE
            comp_data(const_mod_loc) := stat_num;
```

42

```
                    --- INVOKE COMPONENT PARSER TO FILE CONDITION
                        parse_component(stat_line(c0..c1), ct, comp_data(comp_loc+1), cs);
                    --- COMPLAIN IF COMPONENT NOT OF SINGULAR NUMERIC TYPE
                        if cs > 1 or not (ct in numeric_comps or ct = unknown_comp) then
                            cuss (stat_needs_numeric_single, stat_line(c0..c1));
                        end if;
                    end if;


            -----------------------------------------------------------
            --- OTHERWISE                                          ---
            ---    comp_data(comp_loc)      =  otherwise_statement ---
            ---    comp_data(comp_loc+1)    =  end line            ---
            -----------------------------------------------------------

                when otherwise_statement => null;

                --- COMPLAIN IF NO WHEN OPEN...
                    if const_type_save /= when_statement then
                        cuss (otherwise_outside);

                --- OTHERWISE...
                    else
                    --- COMPLAIN IF THERE IS NO "BEFORE" OR "WITHIN"
                        if comp_data(const_mod_loc) = const_num_save then
                            cuss (otherwise_meaningless);
                        end if;
                    --- COMPLAIN IF THERE IS ALREADY AN "OTHERWISE"
                        if comp_data(const_oth_loc) > 0 then
                            cuss (otherwise_already);
                        end if;
                    --- SET "OTHERWISE" LINE
                        comp_data(const_oth_loc) := stat_num;
                    --- RESET LOCATION FOR "END" LINE
                        const_end_loc := comp_loc + 1;
                    end if;

                --- COMPLAIN IF THERE IS EXTRANEOUS MATERIAL ON LINE
                    cuss_extraneous_material (word_break(1, stat_line), stat_line);


            -----------------------------------------------------------
            --- ELSEIF                                              ---
            ---    comp_data(comp_loc)      =  elseif_statement     ---
            ---    comp_data(comp_loc+1)    =  end_line             ---
            ---    comp_data(comp_loc+2)    =  loc of singular boolean component ---
            -----------------------------------------------------------

                when elseif_statement => null;

                --- COMPLAIN IF NO 'IF' OPEN
                    if const_type_save /= if_statement then
                        cuss (else_outside);

                --- OTHERWISE
                    else
                    --- SET "ELSEIF" LINE
                        comp_data(const_mod_loc) := stat_num;
                    --- RESET LOCATION FOR "END" LINE
                        const_end_loc := comp_loc + 1;
                    --- COMPLAIN IF THERE IS ALREADY AN "ELSE"
                        if const_else_num > 0 then
                            cuss (else_already, char(const_else_num));
                        end if;
                    --- SET FLAG TO INDICATE AN "ELSE"
                        const_else_num := stat_num;
                    --- INVOKE COMPONENT PARSER TO FILE CONDITION
                        parse_component(stat_line(c0..c1), ct, comp_data(comp_loc+2), cs);
                    --- COMPLAIN IF COMPONENT NOT OF SINGULAR BOOLEAN TYPE
                        if cs > 1 or not (ct in boolean_comps or ct = unknown_comp) then
                            cuss (stat_needs_boolean_single, stat_line(c0..c1));
                        end if;
                    end if;


            -----------------------------------------------------------
            --- ELSE                                               ---
            ---    comp_data(comp_loc)      =  else_statement      ---
            ---    comp_data(comp_loc+1)    =  end_line            ---
            -----------------------------------------------------------

                when else_statement => null;

                --- COMPLAIN IF NO 'IF' OPEN
                    if const_type_save /= if_statement then
                        cuss (elseif_outside);

                --- OTHERWISE
                    else
                    --- SET "ELSE" LINE
                        comp_data(const_mod_loc) := stat_num;
                    --- RESET LOCATION FOR "END" LINE
                        const_end_loc := comp_loc + 1;
                    --- COMPLAIN IF THERE IS ALREADY AN "ELSE"
                        if const_else_num > 0 then
                            cuss (else_already, char(const_else_num));
                        end if;
                    --- SET FLAG TO INDICATE AN "ELSE"
                        const_else_num := stat_num;
```

43

```
                    end if;

            --- COMPLAIN IF THERE IS EXTRANEOUS MATERIAL ON LINE
                cuss_extraneous_material (word_break(1, stat_line), stat_line);


         _____
         --- END                                                  ---
         ---   comp_data(comp_loc)    =  end_statement              ---
         ---   comp_data(comp_loc+1)  =  pointer to corresponding opener line ---
         _____

            when end_statement =>

            --- IF NO CONSTRUCT IS OPEN...
                if const_loc_save = 0 then
                --- COMPLAIN THAT NO CONSTRUCT OPEN...
                    cuss (no_construct_open);

            --- OTHERWISE...
                else
                --- SAVE POINTER BACK TO CONSTRUCT OPENER
                    comp_data(comp_loc+1) := const_num_save;
                --- COMPLAIN IF TYPE MISMATCH
                    if wird(2, stat_line) /= "" and
                        wird(2, stat_line) /= keyword(const_type_save) then
                        cuss (end_mismatched, wird(2, stat_line) & "  /= " &
                            keyword(const_type_save));
                    end if;
                --- SET LOCATION OF END LINE
                    comp_data(const_end_loc) := stat_num;
                --- ZERO SAVED CONSTRUCT LOCATION TO INDICATE CLOSURE
                    const_loc_save := 0;
                end if;

            --- COMPLAIN IF THERE IS EXTRANEOUS MATERIAL
                cuss_extraneous_material (word_break(2, stat_line), stat_line);


         _____
         --- WAIT                                                  ---
         ---   comp_data(comp_loc)    =  wait_statement             ---
         ---   comp_data(comp_loc+1)  =  loc of singular numeric component ---
         _____

            when wait_statement =>

            --- INVOKE COMPONENT PARSER TO FILE CONDITION
                parse_component(stat_line(c0..c1), ct, comp_data(comp_loc+1), cs);

            --- COMPLAIN IF COMPONENT NOT OF SINGULAR NUMERIC TYPE
                if cs > 1 or not (ct in numeric_comps or ct = unknown_comp) then
                    cuss (stat_needs_numeric_single, stat_line(c0..c1));
                end if;


         _____
         --- CALL                                                  ---
         ---   comp_data(comp_loc)    =  start_statement            ---
         ---   comp_data(comp_loc+1)  =  pointer to subseq block     ---
         _____

            when call_statement =>

            --- FILE INFO ABOUT REFERENCED SUBSEQ
                n_ss_ops := n_ss_ops + 1;
                if n_ss_ops > nssop then
                    cuss (too_many_ss_ops);
                else
                    ss_op_name(n_ss_ops) := pad(wird(2, stat_line), max_name_length);
                    ss_op_stat(n_ss_ops) := stat_num;
                    ss_op_block_loc(n_ss_ops) := comp_loc + 1;
                end if;

        end case;


    _____
    ------- ACTION STATEMENTS ------
    _____

        elsif stat_type in action_statements then

        --- MATERIAL PARTICULAR TO SPECIFIC ACTION STATEMENTS
            case action_statements'(stat_type) is

         _____
         --- SET                                                  ---
         ---   comp_data(comp_loc)    =  set_statement              ---
         ---   comp_data(comp_loc+1)  =  component to be written into ---
         ---   comp_data(comp_loc+2)  =  material to be written      ---
         _____

            when SET_STATEMENT =>

            --- LOCATE "=" OR ":=" OR THE WORD "TO"...
                locate (" TO ", stat_line(c0..c1), cop0, cop1, outside_parens);
                if cop0 = 0 then
                    locate (" := ", stat_line(c0..c1), cop0, cop1, outside_parens);
```

44.

```
                    end if;
                    if cop0 = 0 then
                        locate (" = ", stat_line(c0..c1), cop0, cop1, outside_parens);
                    end if;

            --- COMPLAIN IF NO DELIMITER
                    if cop0 = 0 then
                        cuss (set_delimiter_missing, stat_line(c0..c1));

            --- OTHERWISE...
                    else
                    --- INVOKE COMPONENT PARSER TO FILE COMPONENT
                        parse_component(stat_line(c0..cop0-1), ct_left, cp_left, cs_left, write);
                    --- RECORD THE VARIABLE TO BE LOADED
                        comp_data(comp_loc+1) := cp_left;
                    --- COMPLAIN IF COMPONENT IS NOT OF A TYPE THAT MAY BE SET
                        cuss_if_not_setable (cp_left, stat_line(c0..cop0-1));
                    --- COMPLAIN IF NO LOAD DATA
                        if cop1 >= c1 then
                            cuss (set_data_missing);
                    --- OTHERWISE...
                        else
                        --- INVOKE COMPONENT PARSER TO FILE LOAD MATERIAL
                            parse_component(stat_line(cop1+1..c1), ct_right, cp_right, cs_right, read);
                        --- RECORD THE MATERIAL TO BE LOADED
                            comp_data(comp_loc+2) := cp_right;
                        --- COMPLAIN IF MATERIAL TYPE DISAGREES WITH VARIABLE...
                            if ct_left /= unknown_comp and ct_right /= unknown_comp then
                                if ct_left in boolean_comps and ct_right not in boolean_comps then
                                    cuss (set_data_not_boolean, stat_line(cop1+1..c1));
                                elsif ct_left in numeric_comps and ct_right not in numeric_comps then
                                    cuss (set_data_not_numeric, stat_line(cop1+1..c1));
                                elsif ct_left in character_comps and ct_right not in character_comps then
                                    cuss (set_data_not_character, stat_line(cop1+1..c1));
                                end if;
                            end if;
                        --- COMPLAIN IF MATERIAL IS NOT EQUAL IN SIZE TO VARIABLE, OR SINGULAR...
                            if cs_right /= cs_left and cs_right /= 1 then
                                cuss (set_sizes_incompatible);
                            end if;
                        end if;
                    end if;
```

```
            -------------------------------------------------------------
            -- START / STOP / RESUME                                 ---
            --    comp_data(comp_loc)      := start_statement  or    ---
            --                                stop_statement   or    ---
            --                                resume_statement       ---
            ---   comp_data(comp_loc+1)    -  pointer to seq block    ---
            -------------------------------------------------------------

                when start_statement..resume_statement ->

                --- FILE INFO ABOUT REFERENCED SEQ
                    n_ss_ops := n_ss_ops + 1;
                    if n_ss_ops > nssop then
                        cuss (too_many_ss_ops);
                    else
                        ss_op_name(n_ss_ops) := pad(wird(2, stat_line), max_name_length);
                        ss_op_stat(n_ss_ops) := stat_num;
                        ss_op_block_loc(n_ss_ops) := comp_loc + 1;
                    end if;
```

```
            -------------------------------------------------------------
            -- MESSAGE                                                ---
            --    comp_data(comp_loc)      -  message_statement        ---
            --    comp_data(comp_loc+1)    -  pointer to char string component ---
            -------------------------------------------------------------

                when message_statement ->

                --- PARSE AND FILE COMPONENT
                    parse_component(stat_line(c0..c1), ct, comp_data(comp_loc+1), cs, READ);
                --- COMPLAIN IF COMPONENT NOT CHARACTER STRING
                    if ct not in character_comps then
                        cuss(mess_data_not_character, stat_line(c0..c1));
                    end if;
```

```
            -------------------------------------------------------------
            --- OTHER ACTION STATEMENT TYPES DEFINED BY USER          ---
            -------------------------------------------------------------

                when others =>

                --- PARSE USER-DEFINED ACTION STATEMENT TYPES
                    parse_user_statement(stat_line(c0..c1), stat_type, comp_loc);

            end case;
```

```
        -----------------------------------------
        ------ NON-EXECUTABLE STATEMENTS ------
        -----------------------------------------

        elsif stat_type in nonexecute_statements then

        --- COMPLAIN IF IT'S TOO LATE FOR A DECLARE OR DEFINE
```

45

```
        if defdecs_okay = false then
            cuss (too_late_for_defdec);
        end if;

--- MATERIAL PARTICULAR TO SPECIFIC NON-EXECUTABLE STATEMENTS
    case nonexecute_statements'(stat_type) is

    --------------------------------------------------------------
    --- DECLARE                                                 ---
    ---    comp_data(comp_loc)      =  declare_statement         ---
    --- - - - - - - - - - - - - - - - - - - - - - - - - - - - - ---
    ---    comp_data(comp_loc)      =  xxx_int_var               ---
    ---    comp_data(comp_loc+1)    =  number of pieces (size)   ---
    ---    comp_data(comp_loc+2)    =  pointer to start of name  ---
    ---    comp_data(comp_loc+3)    =  pointer to end of name    ---
    ---    comp_data(comp_loc+4)    =  loc of variable           ---
    --------------------------------------------------------------

        when declare_statement ->

        --- ALLOCATE SPACE (CHANGE LATER IF TYPE NOT BOOLEAN)
            allocate_component(bool_int_var, loc);

        --- FILE DECLARATION NAME (ALWAYS SECOND WORD)
            file_name (wird(2, stat_line), stat_num,
                comp_data(loc+2), comp_data(loc+3));

        --- LOOK FOR THE TYPE KEYWORD "BOOLEAN"
            locate (" BOOLEAN", stat_line(c0..c1), cop0, cop1);
            if cop0 > 0 then
            --- SAVE LOCATION OF INTERNAL VARIABLE
                comp_data(loc+4) := n_bool_int_var + 1;
            else
            --- LOOK FOR THE TYPE KEYWORD "NUMERIC"
                locate (" NUMERIC", stat_line(c0..c1), cop0, cop1);
                if cop0 > 0 then
                --- SAVE LOCATION OF INTERNAL VARIABLE
                    comp_data(loc+4) := n_num_int_var + 1;
                --- OVERWRITE INTERNAL VARIABLE TYPE
                    comp_data(loc) :=
                        half_integer(comp_type_type'pos(num_int_var));
                else
                --- LOOK FOR THE TYPE KEYWORD "CHARACTER"
                    locate (" CHARACTER", stat_line(c0..c1), cop0, cop1);
                    if cop0 > 0 then
                    --- SAVE LOCATION OF INTERNAL VARIABLE
                        comp_data(loc+4) := n_char_int_var + 1;
                    --- OVERWRITE INTERNAL VARIABLE TYPE
                        comp_data(loc) :=
                            half_integer(comp_type_type'pos(char_int_var));
                --- IF NO KEYWORD FOUND, COMPLAIN...
                    else
                        cuss(declare_type_missing);
                    end if;
                end if;
            end if;

        --- LOCATE PARENTHESES, IF ANY
            par0 := location("(", stat_line(c0..c1));
            par1 := location(")", stat_line(c0..c1));
        --- IF THERE ARE PARENTHESES, OBTAIN SIZE OF VARIABLE
            if par1 > par0 then
            --- TRY TO PARSE IT
                parse_component(stat_line(par0..par1), ct, cp, cs);
            --- COMPLAIN IF NOT AN INTEGER LITERAL
                if ct /= num_ntgr_lit then
                    cuss (declare_size_no_good, stat_line(par0..par1));
                    num := 1.0;
                else
                --- EVALUATE LITERAL AND SAVE
                    eval_num_literal (cp, num);
                end if;
            --- SAVE SIZE
                comp_data(loc+1) := half_integer(num);
            --- COMPLAIN IF SIZE MISPLACED
                if par0 < cop0 then
                    cuss (declare_size_misplaced);
                end if;
        --- OTHERWISE ASSUME SIZE IS ONE
            else
                comp_data(loc+1) := 1;
            end if;

        --- DEPENDING ON WHICH TYPE...
            case comp_type_type'val(integer(comp_data(loc))) is
            --- BOOLEAN...
                when bool_int_var ->
                --- INCREMENT BOOLEAN INTERNAL VARIABLE COUNTER
                    n_bool_int_var := n_bool_int_var + comp_data(loc+1);
                --- COMPLAIN IF LIMIT EXCEEDED
                    if n_bool_int_var >= max_bool_int_vars then
                        cuss (too_many_bool_int_vars);
                    end if;
            --- NUMERIC...
                when num_int_var ->
                --- INCREMENT NUMERIC INTERNAL VARIABLE COUNTER
                    n_num_int_var := n_num_int_var + comp_data(loc+1);
                --- COMPLAIN IF LIMIT EXCEEDED
                    if n_num_int_var >= max_num_int_vars then
```

46

```
                                cuss (too_many_num_int_vars);
                            end if;
                    --- CHARACTER...
                        when char_int_var ->
                        --- INCREMENT CHARACTER INTERNAL VARIABLE COUNTER
                            n_char_int_var := n_char_int_var + comp_data(loc+1);
                        --- COMPLAIN IF LIMIT EXCEEDED
                            if n_char_int_var >= max_char_int_vars then
                                cuss (too_many_char_int_vars);
                            end if;
                    --- OTHERS
                        when others => null;
                    end case;


        ----------------------------------------------------------------
        --- DEFINE                                                   ---
        ---    comp_data(comp_loc)      -  DEFINE_STATEMENT          ---
        --- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ---    comp_data(comp_loc)      -  DEFINITION                ---
        ---    comp_data(comp_loc+1)    -  loc of defined component  ---
        ---    comp_data(comp_loc+2)    =  pointer to start of name  ---
        ---    comp_data(comp_loc+3)    -  pointer to end of name    ---
        ----------------------------------------------------------------

            when DEFINE_STATEMENT ->

            --- ALLOCATE SPACE
                allocate_component(DEFINITION, loc);

            --- FILE DEFINITION NAME (ALWAYS SECOND WORD)
                file_name (wird(2, stat_line), stat_num,
                    comp_data(loc+2), comp_data(loc+3));

            --- LOCATE THE WORD "AS"...
                locate (" AS ", stat_line(c0..c1), cop0, cop1, outside_parens);

            --- COMPLAIN IF NO "AS"
                if cop0 - 0 then
                    cuss (no_as_in_definition, stat_line(c0..c1));

            --- OTHERWISE...
                else

                --- INVOKE COMPONENT PARSER TO FILE DEFINED COMPONENT
                    parse_component (stat_line(cop1+1..c1), ct,
                        comp_data(loc+1), cs);

                end if;

        end case;

    end if;


    ------------------------------------
    --- COMPILE-TIME STATEMENTS ---
    ------------------------------------

        else

        case stat_type is

        ------------------------------------------
        --- DIRECT — NO DATA STORED ---
        ------------------------------------------

            when direct_statement =>

            --- SET PRINT LEVEL...
                if wird(2, stat_line) = "PRINT_LEVEL" then
                --- CONVERT THIRD WORD TO A NUMERIC
                    make_numeric (wird(3, stat_line), num, ok);
                --- IF OKAY USE IT TO SET PRINT LEVEL
                    if ok - true then
                        print_level := half_integer(num);
                --- OTHERWISE COMPLAIN
                    else
                        cuss (print_level_not_numeric);
                    end if;

            --- SET SCRIPT NAME...
                elsif wird(2, stat_line) = "SCRIPT_NAME" then
                    script_name := pad(wird(3, stat_line), max_name_length);

            --- SET OPTIMIZATION FLAG...
                elsif wird(2, stat_line) - "OPTIMIZE" then
                    optimize_flag := true;

            --- RESET OPTIMIZATION FLAG...
                elsif wird(2, stat_line) - "NO_OPTIMIZE" then
                    optimize_flag := false;

            --- CHECK DATA BASE FOR ERRORS (IF APPLICABLE)...
                elsif wird(2, stat_line) - "CHECK_DATA_BASE" then
                    CHECK_DATA_BASE;

            --- UNRECOGNIZED...
                else
```

47

```
                        --- COMPLAIN...
                            cuss (unrecognized_directive, wird(2, stat_line));
                        end if;


                    -----------------
                    --- OTHERS ---
                    -----------------

                        when others => null;

                    end case;

            end if;


        ------ LOGIC TO DETECT END OF SCRIPT...

        --- IF THIS IS THE LAST LINE IN BUNDLE, OR IN SCRIPT...
            if n_stats > 0 and (next_type = end_of_input or (stat_type = close_blocker and
               block_type_save = bundle_blocker)) then

            --- MAKE SURE ALL REFERENCED SEQS/SUBSEQS ARE PRESENT
                for i in 1..n_ss_ops loop
                    nb := 0;
                    for j in 1..n_blocks loop
                        if trim(ss_op_name(i)) = block_name(j) then
                            nb := j;
                            comp_data(ss_op_block_loc(i)) := j;
                        end if;
                    end loop;
                --- COMPLAIN IF NOT FOUND
                    if nb = 0 then
                        cuss (seq_subseq_not_found, char(ss_op_stat(i)));
                --- COMPLAIN IF IT SHOULD BE A SUBSEQUENCE
                    elsif block_type(nb) /= subseq_blocker and
                        statement_typ(ss_op_stat(i)) = call_statement then
                            cuss (op_requires_subseq, char(ss_op_stat(i)));
                --- COMPLAIN IF IT SHOULD BE A SEQUENCE
                    elsif block_type(nb) /= seq_blocker and
                        (statement_typ(ss_op_stat(i)) = start_statement or
                         statement_typ(ss_op_stat(i)) = resume_statement or
                         statement_typ(ss_op_stat(i)) = stop_statement) then
                            cuss (op_requires_seq, char(ss_op_stat(i)));
                    end if;
                end loop;

            --- SET DEFAULT SCRIPT NAME
                if trim(script_name) = "" then
                    script_name := pad("SCRIPT", max_name_length);
                end if;

            --- PRINT FILE SUMMARY
                if print_level >= 0 then
                    print_timeliner_usage_summary (trim(script_name));
                end if;

            --- PERHAPS PRINT DATA FILES
                if print_level >= 1 then
                    print_timeliner_data_files (trim(script_name));
                end if;

            --- IF NO CUSSES WRITE OUTPUT FILE
                if n_cuss = 0 then
                    write_data_file ("TL_" & trim(script_name) & ".DATA");
                else
                    n_cuss_total := n_cuss;
                end if;

            --- RESET COUNTERS FOR NEW "BUNDLE" IF ANY...
                n_names   := 0;
                n_blocks  := 0;
                n_stats   := 0;
                n_comps   := 1;
                n_cuss    := 0;
                n_ss_ops  := 0;
                n_bool_int_var := 0;
                n_num_int_var  := 0;
                n_char_int_var := 0;
                n_numeric_lits   := 0;
                n_character_lits := 0;
                trap_max_n_names          := 0;
                trap_max_statement_length := 0;
                trap_max_block_nest_level := 0;
                trap_max_stat_nest_level  := 0;
                trap_max_comp_nest_level  := 0;
                max_bool_buff_usage       := 0;
                max_num_buff_usage        := 0;
                max_char_buff_usage       := 0;

            end if;


        ------ LOGIC TO CHANGE LEVELS UP OR DOWN...

        --- CALL (DOWNWARDS) IF THIS LINE IS OPENER OR MODIFIER
            if (stat_type in block_openers or
                stat_type in construct_openers or
                stat_type in construct_modifiers) and
```

48

```
                    next_type not in construct_modifiers and
                    next_type /= close_blocker and
                    next_type /= end_statement then
            --- DEBUG PRINT
                    if print_level >= 8 then
                        put ("DOWN:  stat_nest_level:  ");
                        put (char(stat_nest_level));
                        put ("         stat_type:  ");
                        put (stat_type);
                        put ("         next_type:  ");
                        put (next_type);
                        new_line;
                    end if;
            --- RECURSIVE INVOCATION
                    parse_statement (return_code);
            --- IF RETURNING TO BLOCK LEVEL, RESET CURRENT BLOCK TYPE
                    if block_type_save in block_openers then
                        current_block_type := block_type_save;
                    end if;
                end if;

            --- EXIT (UPWARDS) IF NEXT LINE IS FINISHER OR MODIFIER...
                if (next_type in construct_modifiers or
                    next_type = close_blocker or
                    next_type = end_statement) and
                    stat_type not in block_openers and
                    stat_type not in construct_openers and
                    stat_type not in construct_modifiers then
            --- DEBUG PRINT
                    if print_level >= 8 then
                        put ("UP:     stat_nest_level:  ");
                        put (char(stat_nest_level));
                        put ("         stat_type:  ");
                        put (stat_type);
                        put ("         next_type:  ");
                        put (next_type);
                        new_line;
                    end if;
            --- EXIT UNLESS ALREADY AT TOP LEVEL
                    if stat_nest_level > 1 then
                        exit;
                    else
                        cuss (too_many_finishers);
                    end if;
                end if;

            --- EXIT IF THIS OR LOWER LEVEL DETECTED END-OF-FILE...
                if next_type = end_of_input then
                    return_code := end_input;
                end if;
                if return_code = end_input then
                    exit;
                end if;

            end loop stat_loop;

        --- DECREMENT NESTING LEVEL
            stat_nest_level := stat_nest_level - 1;

        end parse_statement;


    ---------------------------------------------------------------------
    --- OBTAIN_STATEMENT:  PROCEDURE THAT DOES THE FOLLOWING:          ---
    ---                    *  READS A NEW STATEMENT FROM THE INPUT FILE  ---
    ---                    *  IF IT IS AN "EXECUTABLE" STATEMENT,       ---
    ---                       INCREMENTS n_stats, RESERVES SPACE IN     ---
    ---                       comp_data ARRAY, SETS COMPONENT TYPE IN   ---
    ---                       comp_data ARRAY, AND SETS comp_loc TO     ---
    ---                       POINT TO THE RESERVED AREA IN comp_data   ---
    ---                    *  PRINTS THE STATEMENT WITH INDENTATION     ---
    ---                       DETERMINED BY THE INPUT PARAMETER level   ---
    ---                    *  RETURNS TO THE CALLER THE FOLLOWING INFO: ---
    ---                       *  STRING CONTAINING ENTIRE STATEMENT     ---
    ---                       *  COLUMN LENGTH OF THE STRING            ---
    ---                       *  TYPE OF THE STATEMENT                  ---
    ---                       *  TYPE OF THE NEXT TATEMENT              ---
    ---                       *  SEQUENTIAL STATEMENT NUMBER            ---
    ---                       *  RESERVED LOCATION IN comp_data ARRAY   ---
    ---------------------------------------------------------------------

    procedure obtain_statement (level     : in natural;
                                stat_line : out stat_string_type;
                                stat_leng : out column_type;
                                stat_type : out comp_type_type;
                                next_type : out comp_type_type;
                                stat_num  : out stat_pointer_type;
                                comp_loc  : out comp_pointer_type) is

        -- LOCAL VERSIONS OF OUT PARAMETERS
        statlin : stat_string_type;
        statlen : column_type := 0;
        statype : comp_type_type;
        nextype : comp_type_type;
        statnum : stat_pointer_type := 0;
        comploc : comp_pointer_type := 0;
        -- FIRST AND LAST COLUMNS OF RAW INPUT LINE
        colfrst : column_type;
        collast : column_type;
```

49

```
--- FOR COMPUTING INDENTATION
    indent_save : natural := 0;
--- FOR KEEPING TRACK OF QUOTATION MARKS
    squot : boolean := false;
    dquot : boolean := false;

begin

--- IF THIS IS THE FIRST PASS...
    if n_raw_lines = 0 then
    --- BLANK LINE
        line_raw := (1..max_line_length => ' ');
    --- SET TYPE TO INDICATE START OF INPUT
        line_type := start_of_input;
    end if;

--- SET LINE TYPE FOR OUTPUT
    statype := line_type;

--- BLANK STATEMENT
    statlin := (1..MAX_stat_lengTH => ' ');

--- LOOP TO FIND THE REST OF THE STATEMENT
    line_loop: loop

    --- IF IT'S A FUNCTIONAL STATEMENT...
        if line_type in functional_statements then
        --- INCREMENT STATEMENT COUNTER (CUSS IF NO ROOM)
            if n_stats < max_stats then
                n_stats := n_stats + 1;
                statnum := n_stats;
            else
                cuss (too_many_stats);
            end if;
        --- ALLOCATE SPACE FOR STATEMENT
            allocate_component (line_type, comploc);
        --- SET POINTER TO COMPONENT DATA
            stat_loc(n_stats) := comploc;
        --- PRINT LINE NUMBER
            put (char(n_stats));
        --- SET INDENT ACCORDING TO NESTING LEVEL
            indent_save := indent_reset + indent_delta * (level - 1);
            if line_type in construct_modifiers then
                indent_save := indent_save + indent_delta / 2;
            end if;
            set_col (positive_count(indent_save));

    --- IF IT'S A STATEMENT CONTINUATION...
        elsif line_type = unknown_line then
        --- INDENT TO FIRST WORD BREAK
            set_col (positive_count(indent_save + word_break(1, trim(statlin))+1));

    --- OTHERWISE...
        else
        --- NO INDENTATION
            set_col (positive_count(indent_reset));
        end if;

    --- PRINT LINE
        put (trim(line_raw));
        new_line;

    --- SET FIRST AND LAST COLUMN OF RAW LINE
        colfrst := trim(line_raw)'first;
        collast := trim(line_raw)'last;

    --- RESET LAST COLUMN IF THERE'S A COMMENT
        if location("--", line_raw) > 0 then
            collast := location("--", line_raw) - 1;
        end if;

    --- ADD LINE TO STATEMENT, IF POSSIBLE...
        if statlen + collast - colfrst < MAX_stat_lengTH then

        --- COPY CHARACTER BY CHARACTER...
            for i in colfrst..collast loop
            --- SINGLE OR DOUBLE QUOTE?
                if line_raw(i) = '"' and squot = false then
                    dquot := not dquot;
                elsif line_raw(i) = ''' and dquot = false then
                    squot := not squot;
                end if;
            --  if line_raw(i) = '"' then
            --      dquot := not dquot;
            --  elsif line_raw(i) = ''' then
            --      squot := not squot;
            --  end if;
            --- IF WITHIN QUOTES...
                if squot = true or dquot = true then
                --- COPY AS IS
                    statlen := statlen + 1;
                    statlin(statlen) := line_raw(i);
            --- OTHERWISE...
                else
                --- ELIMINATE MULTIPLE BLANKS, REPLACE TAB AND
                --- CARRIAGE RETURN WITH BLANKS, AND CONVERT TO UPPER-CASE
                    if line_raw(i..i+1) /= "  " then
                        statlen := statlen + 1;
                        if line_raw(i) = ascii.ht or line_raw(i) = ascii.cr then
```

```
                                statlin(statlen) := ' ';
                        else
                                statlin(statlen) := upcase(line_raw(i));
                        end if;
                end if;
            end if;
        end loop;
--- INSERT A BLANK
        statlen := statlen + 1;
        statlin(statlen) :- ' ';

--- OTHERWISE COMPLAIN...
    else
        cuss (statement_too_long);
    end if;

--- READ NEW LINE AND INDICATE IF END-OF-FILE
    line_raw  := (1..max_line_length => ' ');
    line_type := unknown_line;
    begin
        n_raw_lines := n_raw_lines + 1;
        get_line(line_raw, line_leng);
        exception
        when end_error ->
            line_type := end_of_input;
    end;

--- ASCERTAIN TYPE OF NEW LINE
    if line_type /= end_of_input then
            if print_level >= 10 then
                put_line ("obtain_statement calling statement_typ, with:");
                put_line ("       >" & upcase(line_raw) & '<');
            end if;
        line_type := statement_typ(upcase(line_raw));
            if print_level >= 10 then
                put ("obtain_statement receives from statement_typ:     ");
                put (line_type);
                new_line;
            end if;
    end if;

--- EXIT IF NEW LINE BEGINS ANOTHER STATEMENT
    exit line_loop when line_type /= unknown_line or
            statype - blank_line or statype - comment_line;

end loop line_loop;

--- COMPLAIN IF QUOTATION MARKS UNBALANCED
    if squot - true or dquot - true then
        cuss (quotes_unbalanced);
    end if;

--- SET NEXT LINE TYPE FOR OUTPUT
    nextype := line_type;

--- DEBUG PRINT
    if print_level >= 7 then
        put ("obtain_statement:");
        new_line;
        put ("     stat_line: >" & statlin(1..statlen) & "< ");
        new_line;
        put ("     stat_leng: " & char(statlen));
        put ("     stat_type: ");
        put (statype);
        put ("     next_type: ");
        put (nextype);
        put ("     stat_num:  ");
        put (char(statnum));
        put ("     comp_loc:  ");
        put (char(comploc));
        new_line;
    end if;

--- TRAP MAXIMUM STATEMENT LENGTH
    if statlen > trap_max_statement_length then
        trap_max_statement_length := statlen;
    end if;

--- SET OUTPUTS
    stat_line := statlin;
    stat_leng := statlen;
    stat_type := statype;
    next_type := nextype;
    stat_num  := statnum;
    comp_loc  := comploc;

end obtain_statement;


end tl_parser;
```

51